Visualization, Design, and Verification of Ada® Tasking Using Timing Diagrams

by

R.F. Vidale*, P.A. Szulewski**, and J.B. Weiss**

ABSTRACT

This paper recommends the use of timing diagrams in the design and testing of multi-task Ada programs. By displaying the task states vs. time, timing diagrams can portray the simultaneous threads of data flow and control which characterize tasking programs. This description of the system's dynamic behavior from conception to testing is a necessary adjunct to other graphical techniques, such as structure charts, which essentially give a static view of the system. A series of steps is recommended which incorporates timing diagrams into the design process. Finally, a description is provided of a prototype Ada Execution Analyzer (AEA) which automates the production of timing diagrams from VAX/Ada debugger output.

1.0 Introduction

Concurrent programming brings another dimension of complexity to the problem of software design and testing. Unlike sequential programming, where functional decomposition allows the designer to concentrate on one module at a time, concurrent programming in Ada requires the coordination of many modules (tasks) executing in parallel. The requirements for task sequencing must be established early in the design, and carried through into the traditional domain of detailed design. An incomplete understanding of the task sequencing requirements or their erroneous implementation is an invitation to disaster.

* Boston University, Boston, MA
**The Charles Stark Draper Laboratory, Inc., Cambridge, MA
® Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.

Most available software development tools and techniques, based on functional decomposition, do not adequately portray time dependency and thus do not help the developer visualize, design, and verify task sequencing. Tasking, as a programming technique, presents opportunities to improve productivity, maintainability and portability, but also introduces the possibility of programming errors unique to tasking. Incorrect design or implementation of tasking will produce unintended task sequencing which at best degrades system performance, at worst results in deadlock, deadness, or starvation.

Within the past three years, a number of object-oriented design methods have been proposed specifically for Ada. See Booch [BOOC83], Buhr [BUHR84], and Cherry [CHER85], for example. These methods all use the structure-chart type of diagram to describe the architecture of an Ada program. With the exception of Buhr, whose diagrams include some temporal notations, these representations are essentially static, and as such are of limited use in visualizing the overall sequencing of task interactions intended for a design. Buhr does make limited use of timing diagrams in his book [BUHR84] to illustrate the rendezvous, but does not include them in the design process.

It is the opinion of the authors that timing diagrams are a necessary adjunct to structure charts and should be used in conjunction with them first to design an Ada tasking program, then later to verify that it is behaving as expected. Tai [TAIK86] has also recognized the value of timing diagrams (rendezvous graphs, in his terminology) for debugging Ada tasking programs but does not advocate their use in the design process.

2.0 Timing Diagrams in Program Development

Timing diagrams are useful to Ada program developers at several phases in the life cycle. Data flow sequencing must be considered during the requirements analysis, preliminary design, detailed design, debugging, and testing. With tasking the proportion of time devoted to design, in relation to implementation, is much greater than for sequential programs. We propose the following steps for multi-task Ada program development for gaining confidence in the design before and after implementation.

1. Visualize objects and data flows using "cloud diagrams" to represent objects in the problem domain. Single threads of data flow can be shown by numbering them in sequence, but multiple, interacting threads are difficult to show.

2. Use preliminary timing diagrams, which do not show directions of calls, to show scenarios of required task interaction. Steps 1 and 2 are problem-domain representations.

3. Define Ada data structures and code and compile global data types.

4. Transform the problem-domain objects into Ada program units and portray these with structure graphs showing caller-callee relationships. Refine the preliminary timing diagrams to show caller-callee relationships with task ready/blocked state information.

5. Code the structure graphs in Ada as program unit specifications.

6. Code control skeletons in the program unit bodies to implement the task interactions visualized in the timing diagrams and annotated structure graphs.

7. Execute the code skeletons and generate a timing diagram.

8. Compare timing diagrams against desired behavior.

9. Revise design as necessary.

10. Complete Detailed Design of program unit bodies.

11. Generate timing diagrams to verify.

3.0 Automated Timing Diagram Generation

Automated support for the timing diagrams described in the preceeding section is not, to these authors' knowledge, publically available, but would require two forms: predictive and actual.

The preliminary timing diagrams would be predictive of the program's behavior. These diagrams would be drawn before any code is written to guide the developer in constructing the first level of task interaction. Successive, actual timing diagrams would be derived by simulating or executing program units and automatically extracting task trace information.

To date, no work has been done to develop automated support for the predictive diagrams, which is still a manual process. It is, however, feasible that a system, using formal specification and an assertion checker, could be developed to support this activity. There has, however been some work done by the authors of this paper in the development of a tool for generating actual timing diagrams of multi-task Ada programs.

4.0 The Ada Execution Analyzer Prototype

The Ada Execution Analyzer (AEA) Prototype has been developed at The Charles Stark Draper Laboratory, Inc. (CSDL), to explicitly show the relationship of time, concurrent operations, and task communication using the timing diagram format for multi-task VAX/VMS Ada programs. The AEA provides the capability to visually monitor the runtime execution of multitask Ada programs developed in the DEC VAX/VMS Ada Development Environment. The AEA is run as an extension to the VAX/VMS Symbolic Debugger, and thus provides all the capabilities of that debugger plus a graphic display of task execution. The AEA produces both an overview timing diagram which shows up to 20 Ada tasks, and a detailed timing diagram which shows up to 5 selected tasks. An example Overview Timing Diagram is shown in Figure 1 and an example Detailed Timing Diagram is shown in Figure 2. The symbology used in both diagrams is defined in Tables 1 and 2.

The impetus for developing the AEA was the inability of the conventional DEC/Ada debugger to provide visibility into concurrent task behavior. Even though multi-task information is available from the debugger, it is not easily converted to a useful format by manual means. It is a time consuming and error prone process.

The AEA provides graphic timing diagrams on demand from a program run, significantly reducing the debugging time for multitask programs. The availability of such a tool make practical the method outlined in Section 2.0.

The AEA Prototype is written in VAX/Ada and was released for internal use at CSDL in December 1985. As a rapid-prototype, the AEA was produced quickly in order to allow users some functionality and the opportunity to suggest enhancements. To date, the AEA has been used to debug some small tasking programs for both real projects and in-house Ada training problems. User acceptance of the tool has been generally favorable and the tool will likely be maintained as a corporate resource.

5.0 Future Extensions

Extensions to the AEA fall into three categories: short-term, medium-term, and long-term. Short-term extensions (within 6 months) will focus on making the current AEA implementation more user friendly and including some options to reduce clutter in the diagrams by selectively blanking tasks from the diagram.

Medium-term extensions (within 18 months) will focus on transporting the AEA to an embedded microprocessor development environment in order to extract timing diagrams from a target processor.

Long-term extensions (beyond 18 months) might include automatic task sequence checking and automatic generation of program unit body control skeletons. These extensions require the use of a formal specification technique like the Task Sequencing Language (TSL) [HELM85] during development.

6.0 Conclusions

Ada tasking adds a new dimension of complexity which is hard to visualize using established graphical design methods. With this added complexity, it is essential to work out the required task sequencing early in the design and have a means for verifying task sequencing behavior during testing.

Timing diagrams are a natural, easily understood means of visualizing task sequencing in the conceptual and testing phases of concurrent program development. Timing diagrams can evolve with the data-flow picture of a system. They can show time explicitly and can illustrate multiple threads of control including the effects of time slicing. In this manner they can be used to identify serious tasking errors like deadlock, race conditions, and starvation.

A prototype Ada Execution Analyzer, which produces timing diagrams from VAX/Ada debugger output, has demonstrated the value of timing diagrams in understanding the behavior of an Ada program with multi-tasking. The authors believe that the expanded role for timing diagrams suggested in this paper will result in fewer design errors in multi-tasking applications using Ada.

REFERENCES

[BOOC83] Booch, G., Software Engineering with Ada, Menlo Park, CA, Benjamin/Cummings Publishing Company, 1983.

[BUHR84] Buhr, R.J.A., System Design with Ada, Prentice-Hall, Englewood Cliffs, NJ, 1984.

[CHER85] Cherry, G., and B. Crawford, "The PAMELA Methodology," Thought Tools, Inc., Reston, VA, November 1985.

[TAIK86] Tai, K.C., "A Graphical Notation for Describing Executions of Concurrent Ada Programs," ACM Ada Letters, Vol. VI, No. 1, Jan., Feb. 1986.

[HELM85] Helmbold, D., and D. Luckham, "TSL: Task Sequencing Language," Proc. of the Ada International Conference, Paris, France, May 1985.

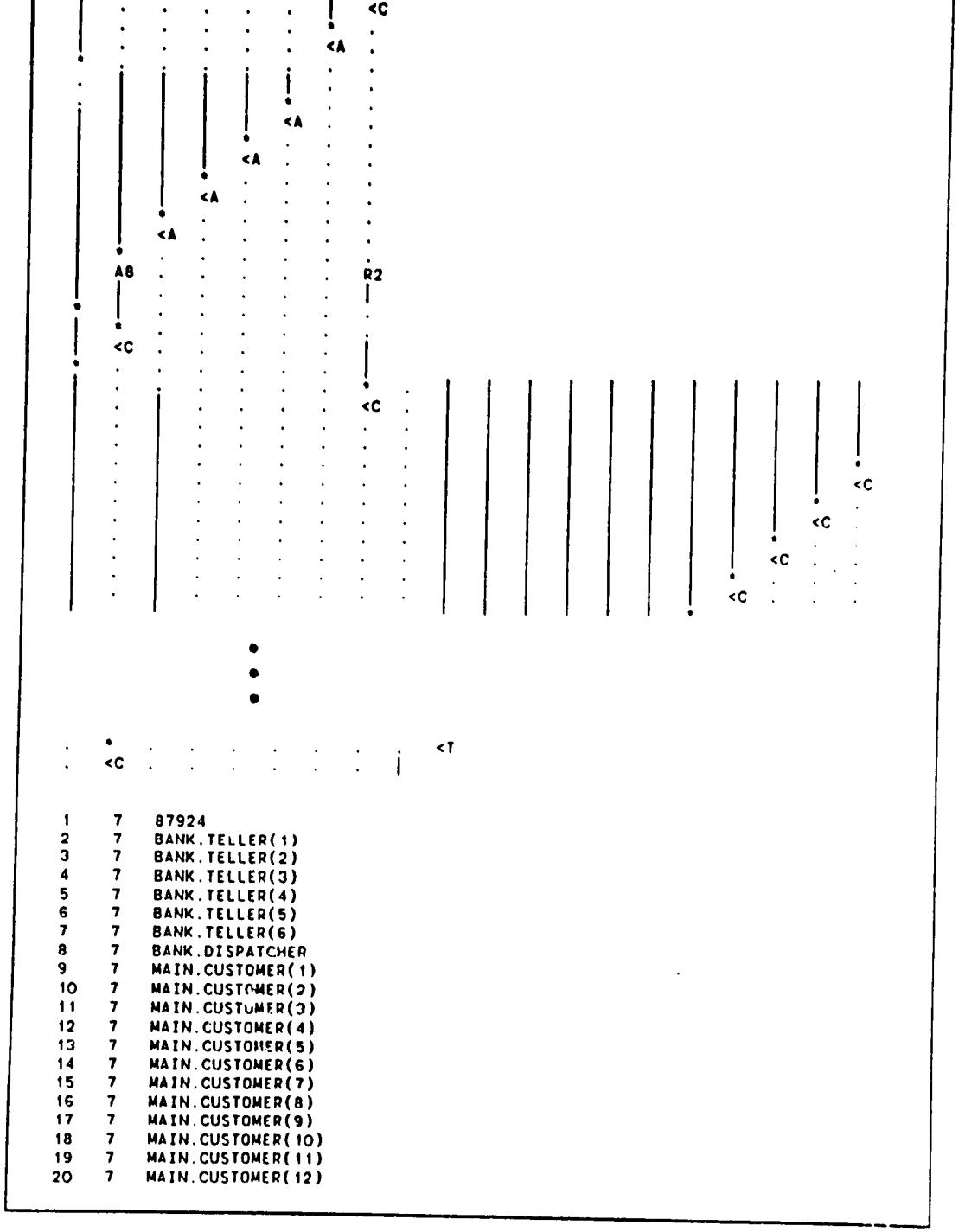

Figure 1. AEA Overview Diagram

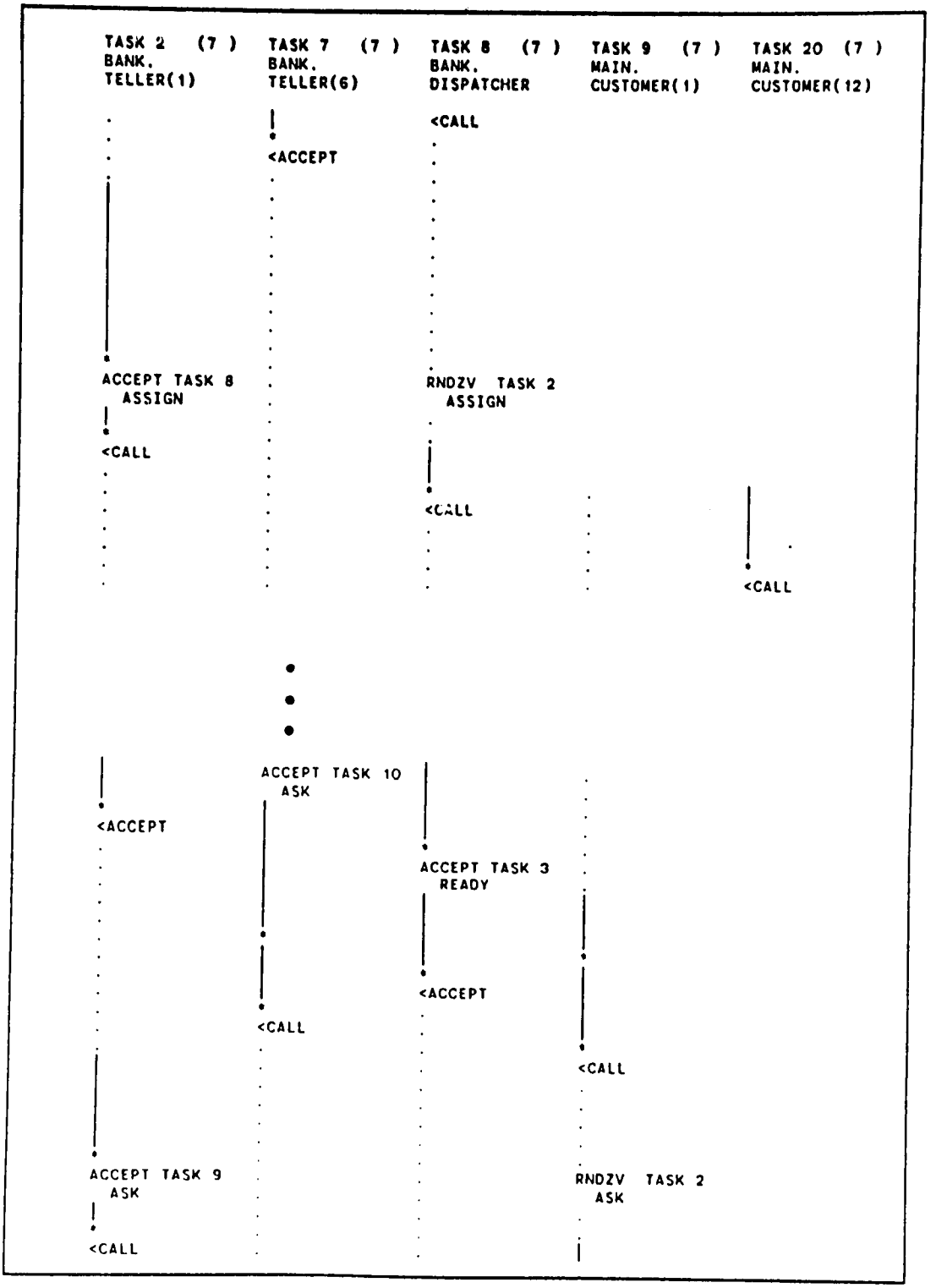

Figure 2. AEA Detailed Timing Diagram

Table 1. AEA Overview and Detailed Diagram States

TASK STATES		
TIMING DIAGRAM SYMBOLS	OVERVIEW DIAGRAM SYMBOLS	MEANING
TASK N (#)	TN P#	Task number N with priority #
UNIT.TASK_NAME		Logical name of program unit that declares TASK_NAME
		POINTS OF RENDEZVOUS:
RNDZV TASK #	R#	Task has rendezvoused with task #
ENTRY_NAME		Task #.ENTRY_NAME
ACCEPT TASK #	A#	Task has accepted call from task #
ENTRY_NAME		Accept ENTRY_NAME
		TASK STATES:
*	*	Task is running
*	*	Task is ready to run
|	|	
•	•	Task is suspended
<TERM	<T	Task has terminated

Table 2. AEA Overview and Detailed Diagram Substates (Part 1 of 2)

TIMING DIAGRAM SYMBOLS	OVERVIEW DIAGRAM SYMBOLS	TASK SUBSTATE	MEANING
<ABORT	<AB	Abnormal	Task has been aborted.
<ACCEPT	<A	Accept	Task is waiting at an accept statement that is not inside a select statement.
<Completed[ab	<CA	Completed[abn]	Task is completed due to an abort statement, but is not yet terminated. In Ada, a task awaiting dependent tasks at its "end" is called "completed". After the dependent tasks are terminated the state changes to terminated.
<Completed[ex	<CE	Completed[exc]	Task is completed due to an unhandled exception, but is not yet terminated. In Ada, a task awaiting dependent tasks at its "end" is called "completed". After the dependent tasks are terminated, the state changes to terminated.
<Completed	<CO	Completed	Task is completed. No abort statement was issued, and no unhandled exception occured.
<Delay	<DL	Delay	Task is waiting at a delay statement.
<Dependents	<DP	Dependents	Task is waiting for dependent tasks to terminate.

Table 2. AEA Overview and Detailed Diagram Substates (Part 2 of 2)

TIMING DIAGRAM SYMBOLS	OVERVIEW DIAGRAM SYMBOLS	TASK SUBSTATE	MEANING
<Dependents[e	<DE	Dependents[exc]	Task is waiting for dependent tasks to allow an unhandled exception to propagate.
<CALL	<C	Entry call	Task is waiting for its entry call to be accepted.
<Invalid State	<IV	Invalid state	There is a bug in the VAX Ada run-time library.
<I/O or AST	<IO	I/O or AST	Task is waiting for I/O completion or some AST. (Asynchronous system true).
<Select or del	<SD	Select or delay	Task is waiting at a select statement with a delay alternative.
<Select or Ter	<ST	Select or term.	Task is waiting at a select statement with a terminate alternative.
<SELECT	<S	Select	Task is waiting at a select statement with neither an else, delay, or terminate alternative.
<Shared resour	<SR	Shared resource	Task is waiting for an internal shared resource.
<Terminated[a	<TA	Terminated[abn]	Task was terminated by an abort.
<Terminated[e	<TE	Terminated[exc]	Task was terminated because of an uhandled exception.
<Terminated	<TN	Terminated	Task terminated normally.
<Timed entry	<TI	Timed entry call	Task is waiting in a timed entry call.